lguest for Plan 9

Ron Minnich

`rminnich@sandia.gov`
Sandia Labs, 2007-6446C

ABSTRACT

We describe the port of Plan 9 to the *lguest* hypervisor. We did this port as part of the creation of the THNX distribution, and to learn *lguest*. Incidentally this port has a few general lessons for people who port Plan 9. This paper follows a narrative structure, in an attempt to show others how a port can be done.

Lguest

Lguest is a fast and simple hypervisor for Linux. It makes no special demands of the hardware, in particular not requiring virtualization support as KVM does. In fact, given the availability of OS sources, it's hard to see a reason NOT to use *lguest*, since one can tailor the OS as needed for best performance, rather than using (e.g.) an emulated NE2000. In the limit, we can even do more Plan 9 style operations, such as mounting the host's TCP stack, rather than going through an emulated network at all.

Methods

We'll start with `l.s`, and work our way up from there. We're starting on a pristine install of a pristine kernel, which will be backed up to `9grid.net` frequently. Would that we had a true version control system; while *yesterday*(1) is nice, file versions are no substitute for a true SCM.

Note that in a few cases, we find a problem, but continue on past it, planning to resolve it later. The reason is that sometimes it is easier to diagnose the problem when more of the system is working.

So, the first step is to

```
cd /sys/src/9; mkdir lguest; dircp pc lguest
```

and make sure it builds and boots. This is a trivial but essential step. We have experimented with modifying files in the `pc` directory, or making a new directory sandbox, and it is simpler to make the new directory.

The information on how to set up the entry code for a guest is in the Linux `.S` entry code. The key points are the startup code, which has an ASCII signature that *lguest*'s loader looks for; and the symbols that demarcate non-interruptible code. How do we get these names into a place that *lguest*'s loader can see them? This detail can be understood from reading the *lguest* loader, found in the Linux kernel source in `Documentation/lguest/lguest.c`.

The *lguest* loader

The *lguest* loader sets up the physical memory much as it would be on a real machine. The loader is located at the top of memory, at the top 4 megabytes of the 4 gigabyte address space; on real machines, the BIOS, APIC controller, and other resources are located at the top of memory. The kernel will be located at `0xc000000`. Unlike many systems, and particularly unlike PCs, the kernel is started with paging enabled and set up. Note that Plan 9 expects *9load* to have turned paging on, but most Linux kernels expect it to be off. There are a few special symbols in the kernel but they are communicated by the guest to the host at startup, via passing a pointer to a structure to the very first hypercall. There is thus no need for the loader to parse the guest symbol table.

Hence, the main issue with understanding the loader requirement is knowing that the loader requires an ELF file, and that the kernel will communicate the required information via *lguest* hypercalls. One additional wrinkle is that the current loader requires that the string `Genuine Lguest` appear somewhere in memory. The loader uses this string to find the first instruction of the guest.

First test: Infinite Loop

Just getting the kernel loaded at all is often a challenge. We can learn a lot by simply asking the loader to load it and then watching the kernel crash and burn. This test highlights an advantage of a virtual machine: the test environment is always operational, and booting an OS is much like starting a program.

So we start by setting `-H5` in the `mkfile` for the *lguest* kernel config. This is a little-known option to *8l* and *ql* that will create a very simple ELF file. We'll just try to do a load and hope we get a complaint from *lguest* that we don't have a true *lguest* kernel.

We also, in deference to the way the *lguest* loader works, put the start at `0x100000` instead of `100020`.

The next problem is an issue with the 8l-generated ELF file. It's a perfectly legal ELF file, but the *lguest* loader does not like it, as the loader expects a Linux ELF file, where the page-aligned segments are also page-aligned in the ELF file — which in turn means the segments can be *mmap*ed instead of copied in. This alignment is not required by ELF, and in fact the *lguest* loader is assuming too much. We made a simple modification (later taken up by Rusty) to `lguest.c` to copy the segments in if the ELF file could not be *mmap*ed. Note that there is a significant performance and space advantage to having the ELF segments page-aligned in the file: they can be mapped copy-on-write. An IBM employee has told us that this page-alignment is one of the reasons he can boot over 6,000 Linux guests on a System 390.

To sum up, the first test is to make sure the *lguest* loader can load the kernel and find the first instruction; and then, jump to our first instruction and loop forever.

`l.s` then looks like this:

```
TEXT _startKADDR(SB), $0
BYTE $0x47; BYTE $0x65; BYTE $0x6e; BYTE $0x75; BYTE $0x69; BYTE $0x6e;
BYTE $0x65; BYTE $0x4c;
BYTE $0x67; BYTE $0x75; BYTE $0x65; BYTE $0x73; BYTE $0x74; BYTE $0
MOVL $_startPADDR(SB), AX
ANDL $~KZERO, AX
JMP* AX
```

and results in this:

```
lguest: unhandled trap 14 at 0x0 (0x0)
```

This may look good, but it it's not: 14 is a GPF. Actually, we've made a few mistakes. The three instructions are actually jumping to low memory, and low memory is not mapped in the page tables (later *lguest* versions will have this mapped, so this sequence would work). As a further test, we put this in:

```
1; JMP 1B
```

Infinite loops are a very handy way to see how far a piece of code is getting. There are, it turns out, a few more problems. In the code above, we null-terminated the string; *lguest* does not use that and is jumping to the 0, and it's not even clear what instructions it is running at that point. So we change the code a bit more:

```
TEXT _startKADDR(SB), $0
BYTE $0x47; BYTE $0x65; BYTE $0x6e; BYTE $0x75; BYTE $0x69; BYTE $0x6e;
BYTE $0x65; BYTE $0x4c;
BYTE $0x67; BYTE $0x75; BYTE $0x65; BYTE $0x73; BYTE $0x74;
TEXT _HACF(SB), $0
MOVL $_HACF(SB), AX
JMP* AX
MOVL $_startPADDR(SB), AX
ANDL $~KZERO, AX
JMP* AX
```

HACF stands for Halt And Catch Fire*. This change works and works well. We can start a guest and have it hang.

Infinite Loop in C

We need to next rewrite `l.s` a bit — it assumes that it is running in low memory, and it's not; it has all the page table setup it needs. So it is time for some surgery. The goal here is to get far enough along so we can get to C and try for a `print`. First, we will try to get to an infinite loop in C: we will put a `while(1);` loop in `main()`. We are changing `l.s` by inserting new code and, for now, leaving the old code there.

Figure 1 shows how `l.s` looks once this work is done.

* Some very old computers literally would burn up given an infinite loop of this type.

```
TEXT _startKADDR(SB), $0 BYTE $0x47; BYTE $0x65; BYTE $0x6e; BYTE $0x75;
BYTE $0x69; BYTE $0x6e;
BYTE $0x65; BYTE $0x4c; BYTE $0x67; BYTE $0x75; BYTE $0x65; BYTE $0x73;
BYTE $0x74
MOVL $MACHADDR, SP
MOVL SP, m(SB) /* initialise global Mach pointer */
MOVL $0, 0(SP) /* initialise m->machno */
ADDL $(MACHSIZE-4), SP /* initialise stack */
/*
 * Need to do one final thing to ensure a clean machine environment,
 * clear the EFLAGS register, which can only be done once there is a stack.
 */
MOVL $0, AX
PUSHL AX
POPFL
PUSHL SI
CALL main(SB)
```

Figure 1: l.s as modified to call main()

This code works, as evidenced by the infinite loop in `main` working.

First Hypercall: crash the kernel

We can start a kernel; can we stop it? The next step on the Xen port was to try for an exit. That was to show we could make hypercalls. On *lguest*, the *exit* call is not a call, really, unlike Xen, so we do not bother. Instead, we need to make the *init* hypercall, but that is run from C, so we have a bit of work to do before we can make further progress.

The next step is to get the basic *lguest* guest data descriptors into `dat.h`. This requires some massaging of source, as the typical Linux usage is to invoke aligned, packed, bit-fields, and other bits of *gcc* that the Plan 9 C compiler does not support.

Once we have the structs, we need to write the hypercall (*hcall*) function. We can use the Xen *hcall* for Plan 9 as a template, and use the `asm()` in `include/linux/lguest.h` to tell us how functions arguments go to registers. The *lguest hcall* is much simpler, however, since all *hcall*s always have four parameters. So, unlike Xen, one function suffices. To create this call, we do the lazy thing: ask the Plan 9 C compiler to create most of the code for us. Create a simple function that takes four arguments, adds them, and returns. Compile that function with `-S`, and take the resulting assembly to produce the hypercall. We show it in Figure 2. The main reason for showing it is to note that we need not save nor restore registers, since *lguest* expects to save them for us, and the function that called us will restore them anyway. This saves a tiny bit of time on the *hcall*.

```
/* lguest hypercall. Always has call # and 3 parameters */
/* these saves of regs are not needed ... */
TEXT hcall(SB), $0
/* PUSHL AX
PUSHL BX
PUSHL CX
PUSHL DX */
MOVL ARG3+12(FP), CX
MOVL ARG2+8(FP), BX
MOVL ARG1+4(FP), DX
MOVL C+0(FP), AX
INT $0x1F
/* POPL DX POPL CX POPL BX POPL AX*/
RET
```

Figure 2: The hypercall function

Recall that our goal here is to set up ONE hypercall to see if anything works. That hypercall is going to be the *init* hypercall. We'll set up an `lguest` structure as in the `lguest.c` code, then do the *hcall*, and see how it goes.

This step went well. We set up the INIT struct, and we did a hypervisor call without having set up the *lguest* data structure, and got a crash as expected. We are on our way. Note that to this point, we have done little but explore different ways to crash the guest. We have the luxury of running in a virtual machine, and can learn quite a bit from the failures. On real hardware, one would probably try to get console output first, and work forward from there. Crashing on normal hardware is only desirable if one has a JTAG or other debugger to make crash analysis easy, and to make restart fast. From this point forward, we are following a track that is a more common porting methodology: get the console going, then get the kernel going.

Test console output

The next important step is to get console I/O working. Life without a console is not really worth living. This step in turn requires quite a bit of setup — we have to get all the `.h` files and the hypercall files to compile, bring in the `lguest_dma` struct, and so on. This bit of time consuming drudgery takes an hour or two but is not really that difficult.

Once we have done this initial work, we can do a test output message. Note at this point there are no real devices. But for further debugging, console output is critical. The test is to drop a direct hypercall into `main()` and see if we get output.

We show `main()` at this intermediate stage in Figure 3.

```
/* empty out the DMA ring for lguest */
for(i = 0; i < LHCALL_RING_SIZE; i++)
    lguest_data.hcall_status[i] = 0xff;
/* before we do anything at all, let's go ahead and talk to host */
hcall(LHCALL_LGUEST_INIT, paddr(&lguest_data), 0, 0);
test.addr[0] = paddr("HI\n");
test.len[0] = 3;
lguest_send_dma(LGUEST_CONSOLE_DMA_KEY, &test);
```

Figure 3: main with console output

This works as expected.

Real console I/O with a real *lguest* device

Once we have the basic interface to *lguest*, we can create the devices to call those functions. The console is quite the easiest. What is nice about *lguest*, as opposed to Xen, is that the I/O interface for all devices is the same. Once we start to get console I/O working, we will have some basic functions for block and Ethernet devices.

There are a number of approaches to providing console services in Plan 9. Our first iteration was to modify `port/devcons.c` to use *lguest*. The consoles on *lguest* are totally unlike those on real hardware. There is no real need to pretend otherwise. To simplify things, we move `port/devcons.c` to `lguest.c` and modify it there. The `port/devcons.c` is not all that portable anyway, at least where hypervisors are concerned. We learned this lesson with Xen. For reasons of space we don't show this file.

The rewrite is fairly basic. We remove all screen references, and just do the hypercall instead of queueing to a non-existent serial device. The next step is to try to get through the rest of `main`. Now that we have more working, we put in a test call to `panic()` in `main`. The hope is that we can run to the `panic` and see a nice test message.

On our next boot we see the following error from *lguest*:

```
lguest: lguest.c:692: Invalid address -266744896
```

One slightly confusing aspect of *lguest* is that it doesn't always print hex numbers. This is really address `F019CBC0`. This type of error almost always means the data segment is not aligned or copied into memory correctly. We can test this with the following code in `main`:

```
static x = 0xdeadbeef;
```

and test the value of `x` later via the following:

```
if (x != 0xdeadbeef)
    hcall(LHCALL_CRASH, paddr("data is not aligned"), 0, 0);
```

This particular check works because it tests a constant in the code segment against one in the data segment. If the data segment is unaligned, the test will fail. It turns out that in this case, the memory is aligned correctly. The turns out the kernel is dying on the `ilock` on `kmesg` in `kmesgputs`. This lock is at `0xf023xxxx`, i.e. in the second 1MB — which raises the possibility of some problem with the page tables set up by *lguest*. What we decide to do, to try to get to a better debugging state, is ignore the problem for now and see if it goes away when we are building our own page tables. This decision may seem odd — should we not try to fix each problem in turn? Sometimes it is best to track a problem down to its ultimate source, but sometimes it is best to get more of the kernel working, since the problem may become easier to figure out given more baseline data.

Disabling `kmesgputs` gets us through to the test panic and `dumpstack` works. Time for `mmuinit` and `trapinit`. In many ports there can be real trouble here. But we now have console I/O for debugging.

Further machine initialization

The `trapinit` was fairly straightforward. Note that the `trapvec` code in the kernel `l.s` is 6 bytes; this code is simply a set of calls with a parameter (the trap number) for each trap. The interrupt descriptor table points to these entries. To build the IDT, the

trapinit code uses this table as a template to form the 8 byte trap vector. As in many instances in Plan 9, a simple, clever piece of code substitutes for *gcc* or *ld* complexity. All we need do in the trapinit loop is add one line:

```
hcall(LHCALL_LOAD_IDT_ENTRY, v, idt[v].d0, idt[v].d1);
```

and we're done.

The code next fails in mmuinit0. We need to add an lgdt call to the mmuinit0; the *lgdt* is traditionally set up in l.s, but in our case, we deferred it.

The cpuidentify() call works acceptably well for bringup. It does identify as GenuineIntel. We decide to defer this issue for later.

We find we are next dying in meminit. This is hardly surprising; meminit does a lot of poking around for legacy PC resources, and even accesses invalid memory addresses looking for the end of memory. This is hardly going to work well in the hypervisor environment, as accesses outside the real memory range will cause a fault.

At this point is makes sense to get parameter passing and E820 right. In this way, we can communicate startup parameters to the kernel. We learned the hard way on Xen that it is far more work to wait to do it later. Once we have parameter passing, we can get confinit right and pass in bits like max memory, and so on. The code to manage the startup is mostly in there anyway.

If you examine the startup code, you can see that we push esi before calling main. This is the passed-in (from *lguest* startup) pointer to the physical address of the parameter block. We change the type signature of main as follows:

```
void main(ulong physboot)
```

and modify main a bit:

```
void *boot;
boot = (void *) kaddr(physboot);
```

This boot parameter can now be used to get the e820 map. There is one bit of oddness from Linux: unlike the standard e820 map, the number of e820 entries is stored elsewhere from the entries themselves.

In the case of *lguest*, there is only one chunk of memory. Also, the Plan 9 meminit code has greatly improved over the years. The meminit code is quite simple as a result, and is shown in Figure 4.

```
static void
lguestscan(void)
{
    ulong flags, base;
    /* Linux standard is to have an entry, not a map, at 2d0 */
    struct e820map *e820map = (struct e820map *)kaddr(E820MAP-4);
    ulong addr, size;
    addr = e820map->map[0].addr;
    size = e820map->map[0].size;
    if (! size)
        panic("memory size is 0");
    map(addr, size, MemRAM);
    /* now we have to map in the last page. This will contain
     * device info.  We don't want it in any memory map, though.
     */
    /* it really is writeable! */
    flags = PTEWRITE|PTEVALID;
    base = size;
    lgd = (void *)(base + KZERO);
    pdbmap(m->pdb, base|flags, (ulong) lgd, BY2PG);
}
```

Figure 4: The meminit code for the Plan 9 guest

Note that last `pdbmap` entry we add. *Lguest* stores paravirtual device info on the last page, and `meminit` is as good a place as any to map it in.

Next we finish up the *lgdt* hypercall, to support the Plan 9 *lgdt*.

The next step is to load our own *pdb* and *cr3* , and those steps work too. Then we hit a real problem with the CPUMACH setup. The `Mach` struct is a per-CPU structure that is mapped at the same virtual address in each CPU. All the structures save the one for CPU 0 are dynamically allocated. One of the structures in the `Mach` struct is the kernel stack for that CPU.

The problem is a kind of chicken-egg type thing. We enter with a linear MMU mapping. The physical address for MACHADDR is set to `paddr(MACHADDR)`, not `paddr(CPU0ADDR)`. When we build page tables with MACHADDR virtual address mapped to CPU0MACH physical address, and call the `putcr3` hypercall, when the actual change happens, our stack disappears and on return we panic and crash. On normal CPUs, this remap step is done before page tables are set up; we don't have that option, as page tables are active on entry. For now, we're going to do the simple thing and leave the linear mapping in, i.e. we're using a MACHADDR virtual address that is not mapped to `paddr(CPU0MACH)`. We are going to worry about this later. We are not sure how to remap the stack page across hypercalls.

This simple change — not remapping MACHADDR — fixed the disappearing stack. The MMU is working to the point that we can start `/386/init`.

First user process — /386/init

The *fork* and *exec* of `/386/init` work. This is a good sign, since it requires creation of the first `proc` structure, with its segments and pages. Also, getting to user code involves a page fault trap. This step also runs successfully.

Once *init* starts, things start to go wrong. It dies while executing code on page 0 — we seem to be taking a page fault in kernel mode, and dying in the trap handler. We have had this problem before on other ports; it seems to be common to initial startup. First

system call is usually an interesting challenge.

The problem turns out to be simple. There is a lot of special *lguest* setup — on the host side, not the guest side — for trap `0x80`, which is the Linux system call, and we had to replicate that setup for `0x40`, the Plan 9 system call. One option is to get Plan 9 to use `0x80` for system call, but the right way to fix this is make the system call number a member of `struct lguest_data`, which we spend some time working on with Rusty. This change, on the Linux side, turns out to be pretty messy, but we finally get to a solution, which we feel is beyond the scope of this paper (since it is Linux that had to change).

Another problem is that we are still occasionally getting invalid DMAs where there is a virtual address on the stack. These are hard to pin down, and we decide to implement a workaround in the hopes of getting more information for debugging. We modify *lguest* to continue even when these errors happen, and as a result we get to a boot prompt (-`boot from`).

It turns out that the *lguest* I/O is almost always asynchronous from the OS. In other words, the guest can set up a DMA, with an on-stack struct; *lguest* may not copy that DMA struct in until later. If the kernel puts pointers to on-stack structures into the DMA, i.e. an on-stack DMA struct, then by the time *lguest* user-mode support code sees the pointer it may be looking at stack variables that are wrong. The fix is simple: make the struct global or static. This change eliminates the problem.

We also realize that on both Xen and *lguest*, modifying the console code was a mistake. We drop our modified console code and create a `uartlg.c` to support the *Lguest* UART. This approach is recommended by Jim McKie.

Traps

The next step is to work out traps. *Lguest* trap numbering is odd. The external traps are 32 and up in the x86. But *lguest* numbers these as 32-relative. So to get interrupt 33, which we assigned to console, we have to ask *lguest* for interrupt 1. However, interrupt 1 is not triggering anything at all in the guest. Resolving this requires a kernel `print`. It turns out the is to do all our interrupts as 32-based, and we had further made a mistake and were not setting interrupts 0-63; we misread a comment in the *lguest* code. In later *lguest*, this code was completely changed and guests just create a full IDT now.

Timers

The next area to deal with is the timers. Did we really get to a single-user prompt without timers? Yes, in fact we did. It is quite amazing how far an OS can get with no timers active. On the Xen port, we were up with a full *rc* before we realized that our timer interrupts were not working at all. Timers have changed a few times in *lguest*; we will only summarize the latest version here. First, we remove all hardware timer code, since it is not needed. While we are at it we get rid of all files starting with `i82`, and remove all reference to them in the `mkfile` and config file. Then we just plug the needed functions into the `devarch.c` file. One problem we stumble over is that the time from *lguest* is divided into a seconds part and a nanoseconds part. But time seems to run at $4\times$ the rate. Weirdly, a `sleep` command will sleep for the right number of seconds, but a `date` command will show time passing far more quickly than it should. The first clue is that it is passing at about $4\times$ the correct rate.

It turns out that the nanosecond timer is actually counting up to 2^{32} nanoseconds,

which is more like 4 seconds. But if we set the fastticks hz to 2^{32}, the timer code panics. The simplest thing to do, it turns out, is cp port/tod.c lguest/lgtod.c, and modify the tod code. Since we know that the timer is always ns, it is simpler to hardcode the choices. The timer code is much simpler and more reliable.

One note on timer interrupts. Plan 9 is a pre-emptive kernel. The only interrupt handlers that can not be interrupted are the two clocks, You can see the test in lguest23/trap.c:363:

```
if(ctl->irq == IrqCLOCK || ctl->irq == IrqTIMER)
    clockintr = 1;
```

We got this wrong the first time on Xen, with the result that we had strange, non-repeatable errors. It's important when you choose interrupt numbers for the clock and timer that they match these values.

Running the second process

Now /386/init is doing our first user-mode *fork*. And, weirdly, it's not working quite right. The symptom here, once we add a print, is that the process faults on virtual address 0x5020 — forever. We've had this problem before, on both Xen and the ARM, and it relates to not properly flushing the TLB when a mapping is changed. It looks like this:

```
if((old&PTEVALID) && lguestpdb)
    flushpg((ulong) &vpt[VPTX(va)]);
```

We had to add this final flush in putmmu or, in some cases, the new mapping would not work.

The second process runs a little further, and starts up boot, and then we see this (note we still have a few printsin the kernel):

```
boot 12: suicide: sys: trap: fault write addr=0x0 pc=0x0000b93d
load stack c04119f0 %
load stack c0413c10
load stack c04119f0
load stack c0413c10
load stack c04119f0
(etc.)
```

This is almost always a corrupted stack, as you switch stacks. Again, we hit this on both Xen and ARM, for different reasons. But a write fault at this stage of the game, and in this manner, can be traced to the fact that the kernel is switching different physical pages under the same virtual address — the stacks run at the same virtual in each process. As we saw, we've had to fix this once already, in putmmu, by flushing the TLB for the new mapping. Our fix here is to flush the stack page each time we switch. It's actually a little odd as these addresses are not colliding. By adding this hypercall to the stack switch code:

```
lazy_hcall(LHCALL_FLUSH_TLB, 1, 0, 0);
```

the problem seems to be fixed.

Useful note: very time we've gotten to multiple processes, and we die with a bad instruction fault at some weird PC in the second process, it's usually stack corruption due to need to flush TLB or stack on context switch.

Devices

The first device we implement is a disk. We will call it `sdlg`, for *storage device lguest*.

Now, once again, we hit the old problem: *9load* sets partitions for the kernel to find. But *9load* has not run. So what should we do? After another *9fans* discussion, we go back to what we know works: we modify `boot/boot.c` to *exec* a script called `diskpart`, which looks like this:

```
/boot/echo "diskpart here ready to serve"
/boot/fdisk -p '#S/sd00/data'
/boot/fdisk -p '#S/sd00/data' > '#S/sd00/ctl'
/boot/prep -p '#S/sd00/plan9'
/boot/prep -p '#S/sd00/plan9' > '#S/sd00/ctl'
/boot/ls -l '#S/sd00'
/boot/echo "diskpart ends"
```

There is not much to the driver, and you can see it in `lguest23/sdlg.c`. It's actually quite easy, and works on almost the first try.

Network

Now we need a network. We start with the Xen network driver as a model, and work from there. Again, the *lguest* I/O interface is so easy that the driver is up almost immediately.

Debugging

At this point we are booting, and the real work of porting begins. Porting is an exercise in orders of magnitude. First come the immediate errors, then the one in ten, and at some point the one in a million errors. The port is never error-free. And, as it happens, in Linux 2.6.24 the I/O model for *lguest* will completely change, and we will need to repeat the I/O-driver-writing process.

We did have one very nasty problem with lost packets. Weirdly enough, we could make the problem go away by running *snoopy*. We never did figure out why this worked, but we did work out the problem.

It turns out the issue was that *lguest* does not hold off on back-to-back packets, and we were not clearing the packet out of the way in time (or at all) — we were using a fixed-area for the incoming packets and copying them out to `Block` structs. We made a huge improvement that solved the problem and in the process created a zero-copy interface. We set up an array of `lguest_dma` structs and a corresponding array of `Block` structs. As an `lguest_dma` struct is filled in by *lguest*, we take the `Block` away and allocate a new one, and send that block on up the chain (`ipiput`). This changes simplifies the driver, makes it much easier to read, reduces the amount of code, and results in having a zero-copy interface.

Pending issues

The port is out there and working, but there are some remaining issues. We are leaving these unfinished since the I/O model is going to change completely as *lguest* cuts over to the new *virtio* interface. These issues and their fixes are:

- No dynamic interrupt allocation. We will fix this with the new *virtio* code.
- no dynamic device scan — Plan 9 stops at first of each type. Again, we intend to wait for the *virtio* code for this fix.

- console *serialoq* still broken. We still don't quite understand this one.
- disk i/o still unoptimized — we are still doing I/O on 4096-byte blocks.
- The whole `cpu0mach` and `mach0` mapping is still not fixed. This problem will go away now that *lguest* supports starting up guests in low memory.

Conclusions

We show, in a narrative style, how we ported Plan 9 to the *lguest* hypervisor. But in a larger sense we hope to have shown a process by which other ports can be performed. Porting Plan 9 is not hard. This port actually took three weeks from start to functioning as a CPU server.

Thanks to Jim McKie for helping out with simple questions, and to Rusty Russell for writing *lguest*.